Étape 8 : fourmis et fourmilières, partie 2.

Jamila Sam & Jean-Cédric Chappelier, 2018

Version : 1.0

But

Terminer les fourmilières et comprendre le concept du « double dispatch ».

Description

Le moment est venu de programmer la méthode permettant aux fourmilières d'évoluer au cours du temps.

Remarque :
Le contenu de cette étape est assez conséquent, notamment en terme de lecture et d'effort de compréhension. Néanmoins, le travail de la prochaine étape sera allégé.

Implémentation

Fourmilière : évolution au cours du temps

Vous ferez en sorte qu'à chaque fois qu'un temps `ANTHILL_SPAWN_DELAY` s'est écoulé, la fourmilière génère (« fait sortir ») une fourmi.

L'algorithme est simplement le suivant :

- tirer une valeur uniformément distribuée entre 0 et 1 à l'aide de l'outil `UniformDistribution` fourni ;
- si cette valeur est inférieure ou égale à la probabilité de voir sortir une ouvrière (attribut de la fourmilière), on crée une ouvrière à l'endroit où se trouve la fourmilière ;
- sinon, on crée une fourmi soldate (toujours à l'endroit de la fourmilière).

Comme pour `FoodGenerator`, il faudra boucler sur cet algorithme tant que le compteur interne est supérieure ou égale à `ANTHILL_SPAWN_DELAY` (revoir si nécessaire l'étape 3).

L'évolution de la fourmilière sera évidemment être pris en charge par une méthode `update` paramétrée par la vue qu'a la fourmilière sur l'environnement (`AnthillEnvironmentView`), de façon analogue à ce que vous avez fait dans les classes `FoodGenerator` ou `Animal`.

Redéfinition & surcharge des méthodes

Nous aimerions modifier le comportement de la classe `AntWorker` pour qu'elle puisse être capable de rapporter de la nourriture. Une idée naturelle serait de redéfinir la méthode `update` de la classe de base `Animal`. Nous aurions donc quelque chose comme :

```
public final class AntWorker extends Ant {
    /*
     * L'annotation `@Override` permet d'indiquer que l'on
     * définit (ou redéfinit) une méthode abstraite (ou déjà
     * définie mais non finale) dans une super-classe.
     * Elle n'est pas nécessaire mais permet d'éviter
     * de bien étranges comportements comme vous allez
     * le constater par la suite...
     */
    @Override
    void update(AnimalEnvironmentView env, Time dt) {
        ...
        Food food = env.getClosestFoodForAnt(getPosition()); // ne compilera pas
        ...
    }
}
```

Cependant, la vue qu'a un animal de base ne permet pas d'obtenir les sources de nourriture.
Nous serions donc tentés d'écrire :

```
public final class AntWorker extends Ant {

    /*
     * Ce code ne compilera pas car aucune super-classe ne
     * définit une méthode ayant cette signature.
     */
    @Override
    void update(AntWorkerEnvironmentView env, Time dt) {
        ...
        Food food = env.getClosestFoodForAnt(getPosition());
```

```
        ...
    }
}
```

Malheureusement, ce code ne compilera pas. En effet, l'annotation `@Override` provoque une erreur de compilation car nous avons essayé de redéfinir une méthode qui n'existe pas dans la super-classe. En effet, `update(AntWorkerEnvironmentView env, Time dt)` et `update(AnimalEnvironmentView env, Time dt)` sont deux méthodes *différentes* même si elles ont des entêtes très proches.

On pourrait alors imaginer qu'il suffit de retirer le `@Override` pour que « tout aille bien » puisqu'en effet, en retirant cette annotation, l'erreur de compilation disparaît. Cependant, ce n'est pas le comportement que l'on souhaite ; le code suivant illustre pourquoi :

```
for (Animal animal : animals) {
    animal.update(this, dt); // appelle update(AnimalEnvironmentView, Time)
                             // de Animal et non pas celui de Ant !
}
```

Même si le type dynamique de `animal` est `AntWorker`, c'est la méthode de `Animal` qui est appelée ! En effet, en Java (comme dans bien d'autres languages de programmation orienté objet), les méthodes ne sont pas polymorphiques sur les arguments, mais uniquement sur l'instance courante (`this`).

L'idée la plus évidente pour contourner ce fait serait d'écrire quelque chose comme :

```
for (Animal animal : animals) {
    // si animal est de type AntWorker, alors on appelle cette méthode-là
    // sinon si animal est de type AntSoldier, alors on appelle cette méthode-ci
}
```

Cependant, cette approche a recours à des *tests de types*, ce qui va a l'encontre de la généricité à laquelle le polymorphisme nous a habitué ! En effet, cela rendrait le code difficile à maintenir (imaginez les changements à faire à chaque introduction d'un niveau type d'animal !).

Pour contourner ce problème, il est possible d'utiliser la technique dite du « *double dispatch* ». Pour préparer cela, définissez les méthodes protégées suivantes :

- `void seekForFood(AntWorkerEnvironmentView env, Time dt)` dans `AntWorker` ; cette méthode est chargée de modéliser le comportement spécifique de la fourmi ouvrière, c.-à-d. rechercher de la nourriture ;
- `void seekForEnemies(AntEnvironmentView env, Time dt)` dans `AntSoldier` ; cette méthode permet à la fourmi soldate de chercher et combattre un ennemi.

Pour le moment, ces deux méthodes ne feront simplement qu'appeler `move()`. Ces méthodes ne seront pas appelées directement dans `update()`, mais uniquement de manière indirecte, comme vous allez le voir par la suite.

Double dispatch

Il est possible d'éviter les tests de types sur des arguments de méthodes en ayant recours à la technique du « *double dispatch* » : on contourne le fait que les méthodes en Java ne soient pas polymorphiques sur les arguments en utilisant la surcharge. L'idée est la suivante :

- on déclare dans `Animal` la méthode *abstraite* suivante :

  ```
  void specificBehaviorDispatch(AnimalEnvironmentView env, Time dt)
  ```

- ensuite, on déclare dans l'interface `AnimalEnvironmentView` les méthodes

  ```
  void selectSpecificBehaviorDispatch(AntWorker antWorker, Time dt)
  ```

et

```
void selectSpecificBehaviorDispatch(AntSoldier antSoldier, Time dt)
```

Ces méthodes s'occuperont respectivement d'appeler les méthodes qui sont spécifiques à `AntWorker` et à `AntSoldier`, c.-à-d. respectivement `seekForFood()` et `seekForEnemies()`.

La méthode `update()` de la classe `Animal` appellera la méthode `specificBehaviorDispatch()`. Comme `specificBehaviorDispatch()` sera définie dans *chacune* des sous-classes non abstraites, elle permettra de déterminer dynamiquement le type de l'animal en question. Elle appellera alors la méthode

```
void selectSpecificBehaviorDispatch(AntWorker antWorker, Time dt)
```

si l'animal est une fourmi ouvrière ou

```
void selectSpecificBehaviorDispatch(AntSoldier antSoldier, Time dt)
```

si l'animal est une fourmi soldate.
A son tour, `selectSpecificBehaviorDispatch()` pourra appeler les méthodes de comportement de l'animal en connaissance de son type dynamique.

On aura ainsi le shéma d'appel suivant :

```
// Dans Environment :
for (Animal animal : animals) {
    animal.update(this, dt);
}

// Dans Animal :
void update(AnimalEnvironmentView env, Time dt) {
    ...
```

```
    this.specificBehaviorDispatch(env, dt);
    ...
}

// Dans une sous-classe qui hérite de Animal (p.ex. AntWorker) :
@Override
void specificBehaviorDispatch(AnimalEnvironmentView env, Time dt) {
    // A ce moment là, on sait que l'on à affaire à un AntWorker.
    // Grâce à l'appel suivant, on informe AnimalEnvironmentView de notre type !

    env.selectSpecificBehaviorDispatch(this /* ici le type de this est AntWorker ! */, dt);
}

// Dans Environment, qui implémente AnimalEnvironmentView :
@Override
void selectSpecificBehaviorDispatch(AntWorker antworker, Time dt) {
    // Grâce à la surchage des méthode, cette méthode
    // sera appelée par AntWorker dans son specificBehaviorDispatch.


    // Ici, on est libre d'appeler n'importe quelle méthode non privée de
    // AntWorker, et lui passer n'importe quelle paramètre, en particulier
    // une AntWorkerEnvironmentView !

    antworker.seekForFood(this /* AntWorkerEnvironmentView que Environment
                                * implémente */
                          , dt);
}

// Dans AntWorker :
void seekForFood(AntWorkerEnvironmentView env, Time dt) {
    // Enfin !!!!
}
```

Le code pour `AntSoldier` est similaire.

Le mécanisme général décrit ici s'appelle « **double** dispatch » car il généralise le polymorphisme à des relations à *deux* classes : on veut d'un coté un comportement polymorphique de la mise-à-jour chaque entité et de l'autre coté un comportement « polymorphique » de l'environnement en fonction l'entité qui doit le modifier. Ce « double polymorphisme » (d'un coté par les entités et de l'autre par les différentes vues de l'environnement) n'est pas possible directement en Java et nécessite donc la technique de « double dispatch » présentée ici.

Nous faisons le choix délibéré de différer le codage du comportement spécifique des termites. Nous y reviendrons un peu plus tard dans le projet.

Modification de `Animal.update()`

Jusqu'à maintenant, l'algorithme régissant l'évolution des animaux ne gérait que leur déplacement et leur durée de vie. Nous aimerions que la gestion de la durée de vie reste la même pour tous les animaux. En revanche, chaque animal sera libre de décider s'il souhaite se déplacer ou non. Nous allons pour cela retoucher légèrement la méthode `update()` de `Animal`. Jusqu'ici il vous a été demandé d'appeler la méthode `move()` ; il faudra désormais remplacer l'appel de la méthode `move()` par celui de la méthode fraîchement déclarée `specificBehaviorDispatch()` (lequel permettra au final d'appeler la méthode de comportement spécifique aux fourmis soldates et ouvrières comme expliqué plus haut). Ainsi, les sous-classes décideront de la manière dont le déplacement sera géré.

Vous remarquerez que nous souhaiterions empêcher que les sous-classes de `Animal` puisse redéfinir la gestion de la durée de la vie. Un moyen simple est d'empêcher la redéfinition de la méthode `update()`.

Tests et soumission

Tests locaux

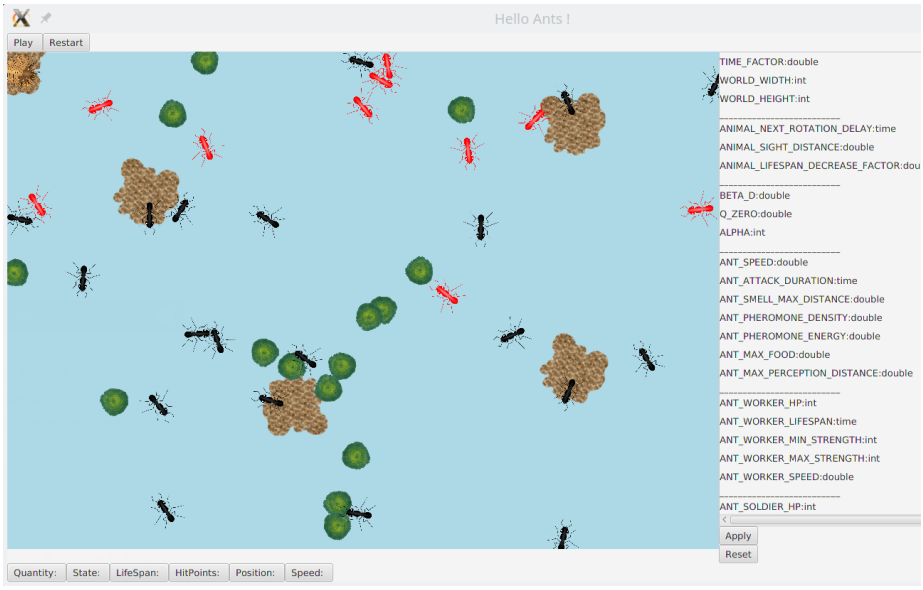

Figure 1: Exemple d'exécution au terme de l'étape 8

Nous vous encourageons, comme toujours à continuer d'ajouter vos propres tests ponctuels dans le fichier `ch/epfl/moocprog/tests/Main.java`.

Parallèlement, pour cette étape, les nouveaux comportements introduits sont faciles à constater visuellement.

Une fois l'application graphique lancée vous devriez voir émerger des fourmis des vos fourmilières (des ouvrières en noir et des soldates en rouge), comme montré par le figure 1.

Les fourmis doivent se déplacer comme les termites à l'étape précédente (avec notamment la possibilité d'augmenter la fréquence des changements de direction via le panneau de configuration à droite).

Vérifiez entre autres :

- qu'il est possible d'augmenter/diminuer la fréquence de création des fourmis au moyen du paramètre `ANTHILL_SPAWN_DELAY` ;
- qu'il est possible de contrôler les proportions de fourmis soldates et ouvrières au moyen du paramètre `ANTHILL_WORKER_PROB_DEFAULT`.

Les termites doivent quand à elles rester figées dans le coin en haut à gauche, ce qui est normal vu qu'on ne les a pas encore dotées explicitement de leur comportement spécifique !

Une petite vidéo d'exemple de simulation obtenue à ce stade est disponible dans le matériel de la semaine 4.

Soumission

Pour soumettre votre devoir au correcteur automatique, il faut créer un fichier ZIP contenant **tous** vos fichiers sources personnels, depuis la racine `ch/` ; ceux de cette étape, mais aussi ceux de l'étape précédente. Concrètement, pour ce devoir ci, votre fichier ZIP doit donc contenir exactement les fichiers suivants :

```
ch/epfl/moocprog/AnimalEnvironmentView.java
ch/epfl/moocprog/Animal.java
ch/epfl/moocprog/AntEnvironmentView.java
ch/epfl/moocprog/AnthillEnvironmentView.java
ch/epfl/moocprog/Anthill.java
ch/epfl/moocprog/Ant.java
ch/epfl/moocprog/AntSoldier.java
ch/epfl/moocprog/AntWorkerEnvironmentView.java
ch/epfl/moocprog/AntWorker.java
ch/epfl/moocprog/Environment.java
ch/epfl/moocprog/FoodGeneratorEnvironmentView.java
ch/epfl/moocprog/FoodGenerator.java
ch/epfl/moocprog/Food.java
```

`ch/epfl/moocprog/Pheromone.java`
`ch/epfl/moocprog/Positionable.java`
`ch/epfl/moocprog/RotationProbability.java`
`ch/epfl/moocprog/Termite.java`
`ch/epfl/moocprog/ToricPosition.java`

Note : si c'est plus pratique pour vous, vous *pouvez* aussi faire un zip qui contient tout le sous-répertoire `ch/epfl/moocprog`, y compris, donc, les fichiers que nous vous avons fournis. Ces fichiers seront simplement ignorés par le correcteur automatique (et remplacés par les nôtres). L'énoncé « *Procédure de soumission* » de la semaine 1 vous indique comment procéder. **Avant de passer à l'étape suivante, n'oubliez pas de faire une sauvegarde de l'étape en cours**, comme indiqué dans le même document.